3.10 Testing Functions II: `hypothesis`

When we introduced if statements in Section 3.4, we discussed how unit tests could be used to perform *white box testing*, where the goal is to "cover" all possible execution paths with unit tests. Unit tests really excel in this scenario because we can determine what the inputs of a function should be to reach a particular branch.

But choosing unit test inputs also imposes challenges on the programmer writing those tests. How do we know we have "enough" inputs? What properties of the inputs should we consider? For example, if our function takes a `list[int]`, how long should our input lists be, should they contain duplicates, and what should the values inside the list be? For each choice of answers to these questions, we then need to choose a specific input and calculate the expected output to write a unit test.

In this section, we introduce a different form of testing called *property-based testing*, using the Python module `hypothesis`. The main advantage of property-based testing with `hypothesis` is that we can write one test case that calls the function being tested *multiple inputs* that the `hypothesis` library chooses for us automatically. Property-based tests are not intended to replace unit tests—both have their role in testing and both are important.

Property-based testing

The kinds of tests we've discussed so far involve defining *input-output pairs*: for each test, we write a specific input to the function we're testing, and then use `assert` statements to verify the correctness of the corresponding output. These tests have the advantage that writing any one individual test is usually straightforward, but the disadvantage that choosing and implementing test cases can be challenging and time-consuming.

There is another way of constructing tests that we will explore here: *property-based testing*, in which a single test typically consists of a large set of possible inputs that is generated in a programmatic way. Such tests have the advantage that it is usually straightforward to cover a broad range of inputs in a short amount of code; but it isn't always easy to specify exactly what the corresponding outputs should be. If we were to write code to compute the correct answer, how would we know that *that* code is correct?

So instead, property-based tests use `assert` statements to check for *properties* that the function being tested should satisfy. In the simplest case, these are properties that every output of the function should satisfy, regardless of what the input was. For example:

- The *type* of the output: "the function `str` should always return a string."
- *Allowed values* of the output: "the function `len` should always return an integer that is greater than or equal to zero."

- *Relationships* between the input and output: “the function max(x, y) should return something that is greater than or equal to both x and y.”
- *Relationships* between two (or more) input-output pairs: "for any two lists of numbers nums1 and nums2, we know that sum(nums1 + nums2) == sum(nums1) + sum(nums2).

These properties may seem a little strange, because they do not capture precisely what each function does; for example, str should not just return any string, but a string that represents its input. This is the trade-off that comes with property-based testing: in exchange for being able to run our code on a much larger range of inputs, we write tests which are imprecise characterizations of the function’s inputs. The challenge, then, with property-based testing is to come up with good properties that narrow down as much as possible the behaviour of the function being tested.

Using hypothesis

As a first example, let’s consider our familiar is_even function, which we define in a file called my_functions.py:[1]

```python
# Suppose we've saved this in my_functions.py

def is_even(value: int) -> bool:
    """Return whether value is divisible by 2.

    >>> is_even(2)
    True
    >>> is_even(17)
    False
    """
    return value % 2 == 0
```

Rather than choosing specific inputs to test is_even on, we’re going to test the following two *properties*:

- is_even always returns True when given an int of the form 2 * x (where x is an int)
- is_even always returns False when given an int of the form 2 * x + 1 (where x is an int)

One of the benefits of our previous study of predicate logic is that we can express both of these properties clearly and unambiguously using symbolic notation:

$$\forall x \in \mathbb{Z},\ \text{is\_even}(2x)$$
$$\forall x \in \mathbb{Z},\ \neg\text{is\_even}(2x + 1)$$

[1] You can follow along in this section by creating your own files!

Now let's see how to express these properties as test cases using hypothesis. First, we create a new file called test_my_functions.py, and include the following "test" function:[2]

```
# In file test_my_functions.py
from my_functions import is_even


def test_is_even_2x(x: int) -> None:
    """Test that is_even returns True when given a number of the form 2*x."""
    assert is_even(2 * x)
```

Note that unlike previous tests we've written, we have not chosen a specific input value for is_even! Instead, our test function test_is_even_2x takes an an integer for x, and calls is_even on 2 * x. This is a more general form of test because now x could be any integer.

So now the question is, how do we actually call test_is_even_2x on many different integer values?[3] This is where hypothesis comes in. In order to generate a range of inputs, the hypothesis module offers a set of *strategies* that we can use. These strategies are able to generate several values of a specific type of input. For example, to generate int data types, we can use the integers strategy. To start, we add these two lines to the top of our test file:

```
# In file test_my_functions.py
from hypothesis import given                    # NEW
from hypothesis.strategies import integers   # NEW

from my_functions import is_even


def test_is_even_2x(x: int) -> None:
    """Test that is_even returns True when given a number of the form 2*x."""
    assert is_even(2 * x)
```

Just importing given and integers isn't enough, of course. We need to somehow "attach" them to our test function so that hypothesis knows to generate integer inputs for the test. To do so, we use a new piece of Python syntax called a **decorator**, which is specified by using the @ symbol with an expression in the line immediately before a function definition. Here is the use of a decorator in action:

```
# In file test_my_functions.py
from hypothesis import given
from hypothesis.strategies import integers
```

[2] Make sure that my_functions.py and test_my_functions.py are in the same directory.

[3] You could run this file in the Python console and call it manually on different arguments, but there must be a better way!

```
from my_functions import is_even


@given(x=integers())  # NEW
def test_is_even_2x(x: int) -> None:
    """Test that is_even returns True when given a number of the form 2*x."""
    assert is_even(2 * x)
```

The line `@given(x=integers())` is a bit tricky, so let's unpack it. First, `integers` is a `hypothesis` function that returns a special data type called a **strategy**, which is what `hypothesis` uses to generate a range of possible inputs. In this case, calling `integers()` returns a strategy that simply generates `int`s.

Second, `given` is a `hypothesis` function that takes in arguments in the form `<param>=<strategy>`, which acts as a mapping for the test parameter name to a strategy that `hypothesis` should use for generating arguments for that parameter.

We say that the line `@given(x=integers())` *decorates* the test function, so that when we run the test function, `hypothesis` will call the test several times, using `int` values for `x` as specified by the strategy `integers()`. Essentially, `@given` helps automate the process of "run the test on different `int` values" for us!

And finally, To actually run the test, we use `pytest`, just like before:

```
# In file test_my_functions.py
from hypothesis import given
from hypothesis.strategies import integers

from my_functions import is_even


@given(x=integers())
def test_is_even_2x(x: int) -> None:
    """Test that is_even returns True when given a number of the form 2*x."""
    assert is_even(2 * x)


if __name__ == '__main__':
    import pytest
    pytest.main(['test_my_functions.py', '-v'])
```

Testing odd values

Just like with unit tests, we can write multiple property-based tests in the same file and have `pytest` run each of them. Here is our final version of `test_my_functions.py` for this example, which adds a second test for numbers of the form $2x + 1$.

```
# In file test_my_functions.py
from hypothesis import given
from hypothesis.strategies import integers

from my_functions import is_even


@given(x=integers())
def test_is_even_2x(x: int) -> None:
    """Test that is_even returns True when given a number of the form 2*x."""
    assert is_even(2 * x)


@given(x=integers())
def test_is_even_2x_plus_1(x: int) -> None:
    """Test that is_even returns False when given a number of the form 2*x +
        1."""
    assert not is_even(2 * x + 1)


if __name__ == '__main__':
    import pytest
    pytest.main(['test_my_functions.py', '-v'])
```

Using `hypothesis` *with collections*

Now let's consider a more complicated example, this time involving lists of integers. Let's add the following function to `my_functions.py`:

```
# In my_functions.py


def num_evens(nums: list[int]) -> int:
    """Return the number of even elements in nums."""
    return len([n for n in nums if is_even(n)])
```

Let's look at one example of a property-based test for `num_evens`. For practice, we'll express this property in predicate logic first. Let $\mathcal{L}_{int}$ be the set of lists of integers. The property we'll express is:

$$\forall \text{nums} \in \mathcal{L}_{\text{int}},\ \forall x \in \mathbb{Z},\ \text{num\_evens}(\text{nums} + [2x]) = \text{num\_evens}(\text{nums}) + 1$$

Translated into English: for any list of integers $nums$ and any integer x, the number of even elements of `nums + [2 * x]` is one more than the number of even elements of `nums`.

We can start using the same idea as our `is_even` example, by writing the test function in `test_my_functions.py`.

```python
# In test_my_functions.py
def test_num_evens_one_more_even(nums: list[int], x: int) -> None:
    """Test num_evens when you add one more even element."""
    assert num_evens(nums + [2 * x]) == num_evens(nums) + 1
```

Now we need to use `@given` again to tell `hypothesis` to generate inputs for this test function. Because this function takes two arguments, we know that we'll need a decorator expression of the form

```python
@given(nums=..., x=...)
```

We can reuse the same `integers()` strategy for `x`, but what about `nums`? Not surprisingly, we can import the `lists` function from `hypothesis.strategies` to create strategies for generating lists! The `lists` function takes in a single argument, which is a strategy for generating the elements of the list. In our example, we can use `lists(integers())` to return a strategy for generating lists of integers.

Here is our full test file (with the `is_even` tests omitted):

```python
# In file test_my_functions.py
from hypothesis import given
from hypothesis.strategies import integers, lists  # NEW lists import

from my_functions import is_even, num_evens


@given(nums=lists(integers()), x=integers())  # NEW given call
def test_num_evens_one_more_even(nums: list[int], x: int) -> None:
    """Test num_evens when you add one more even element."""
    assert num_evens(nums + [2 * x]) == num_evens(nums) + 1


if __name__ == '__main__':
    import pytest
    pytest.main(['test_my_functions.py', '-v'])
```

Choosing "enough" properties

The property test expressed in `test_num_evens_one_more_even` is pretty neat, but it by itself is not sufficient to verify the correctness of the `num_evens` function. For example, this property would also hold true if `num_evens` simply returned the length of the list, rather than the number of even elements.

This is drawback with property-based tests: even though we can now check some property for very many inputs automatically, a single property alone does not guarantee that a function is correct. The ideal goal of property-based testing, then, is *choosing properties to verify*, so that if all of the properties are verified, then the function must be correct. This sounds too good to be true, and it often is—as functions get more complex, it is challenging or even impossible to find such a set of properties.

But for `num_evens`, a relatively simple function, it is actually possible to *formally prove* the following statement, which tells us exactly which properties we need to check.

Theorem (correctness for `num_evens`). An implementation for `num_evens` is correct (i.e., returns the number of even elements for any list of numbers) *if and only if* it satisfies all three of the following:

1. $\text{num\_evens}([]) = 0$
2. $\forall \text{nums} \in \mathcal{L}_{\text{int}},\ \forall x \in \mathbb{Z},\ \text{num\_evens}(\text{nums} + [2x]) = \text{num\_evens}(\text{nums}) + 1$
3. $\forall \text{nums} \in \mathcal{L}_{\text{int}},\ \forall x \in \mathbb{Z},\ \text{num\_evens}(\text{nums} + [2x+1]) = \text{num\_evens}(\text{nums})$

Proving such a statement is beyond the scope of this chapter, but if you're curious it is closely related to the proof technique of *induction*, which we will cover formally later this year. But the actual statement is pretty amazing: it tells us that with just one unit test (for `nums = []`) and two property tests, we can be certain that our `num_evens` function is correct!

CSC110 Course Notes Home